Docket No. EN995139

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In Re Application of

Applicant : G. W. Wilhelm, Jr.

Serial No. : 08/820,181 Group : 2151

Filed : 14 Mar 1997 Examiner : Majid Banankhah

Entitled : System and Method for Queue-less Enforcement of Queue-like Behavior on Multiple Threads Accessing a Scarce Resource [as amended]

Assistant Commissioner For Patents
Washington, D.C. 20231

CERTIFICATE OF MAILING UNDER 37 CFR 1.8(a)

I hereby certify that the following attached correspondence comprising:

Acknowledgment Postcard
Certificate of Mailing
Transmittal of Appeal Brief (2 copies)
Appeal Brief with Attachments (3 copies)

is being deposited with the United States Postal Service as first class mail in an envelope addressed to:

Commissioner of Patents and Trademarks
Washington, D. C. 20231

on 24 Sep 2001.
(date)

JUDITH A. BECKSTRAND
(Name of person mailing paper or fee)

Judith Anne Beckstrand
(Signature of person mailing paper or fee)

a:\certmail.wpd

Docket No. EN995139 *PATENT*

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re application of: G. W. Wilhelm, Jr.

Application No.: 08/820,181 Group No.: 2151

Filed: 14 Mar 1997 Examiner: M. Banankhah

For: System and Method for Queue-less Enforcement of Queue-like Behavior on Multiple Threads Accessing a Scarce Resource

Assistant Commissioner for Patents
Washington, D.C. 20231

TRANSMITTAL OF APPEAL BRIEF
(PATENT APPLICATION – 37 C.F.R. § 1.192)

1. TRANSMITTAL

Transmitted herewith, in triplicate, is the APPEAL BRIEF in this application, with respect to the Notice of Appeal filed on 25 July 2001.

2. STATUS OF APPLICANT

This application is on behalf of other than a small entity.

3. FEE FOR FILING APPEAL BRIEF

Pursuant to 37 C.F.R. § 1.17(c), the fee for filing the Appeal Brief is $310.00 for other than a small entity.

CERTIFICATE OF MAILING (37 C.F.R. § 1.8(A))

I hereby certify that this correspondence is, on the date shown below, being deposited with the United States Postal Service with sufficient postage as first class mail in an envelope addressed to the Assistant Commissioner for Patents, Washington, D.C. 20231.

Name: Judith A. Beckstrand
Date: 24 Sep 2001 Signature: Judith A. Beckstrand

---

4. EXTENSION OF TERM

The proceedings herein are for a patent application and the provisions of 37 C.F.R. § 1.136 apply. Applicant believes that no extension of term is required. However, if an extension of term is required, please consider this a petition therefor.

5. TOTAL FEE DUE:

The total fee due is:

Appeal brief fee	$ 310.00
Extension fee (if any)	$
Total Fee Due:	$ 310.00

6. FEE PAYMENT:

Charge IBM Deposit Account No. 09-0457 the sum of $ 310.00. A duplicate of this transmittal is attached.

7. FEE DEFICIENCY

This is a request to charge IBM Deposit Account No. 09-0457 for any required additional extension and/or fee, or for any required additional fee for claims.

Shelley M Beckstrand
Attorney for Applicant
Reg. No. 24,886

314 Main Street
Owego, NY 13827

Phone: (607) 687-9913
Fax: (607) 687-7848
Cell phone: (607) 427-0802

#10/Appeal Brief
A.U. 2771. NCBe
10/2/01
2700

Docket No.: EN995139 *PATENT*

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re application of: G. W. Wilhelm, Jr.

Application No.: 08/820,181 Group No.: 2151

Filed: 14 March 1997 Examiner: M. Banahkhah

For: System and Method for Queue-less Enforcement of Queue-like Behavior On Multiple Threads Accessing a Scarce Resource

Assistant Commissioner for Patents
Washington, D.C. 20231

ATTENTION: Board of Patent Appeals and Interferences

APPELLANT'S BRIEF (37 C.F.R. § 1.192)

This brief is in furtherance of the Notice of Appeal, filed in this case on 25 July 2001.

The fees required under 37 C.F.R. §1.17, and any required petition for extension of time for filing this

---

CERTIFICATE OF MAILING (37 C.F.R. § 1.8(A))

I hereby certify that this correspondence is, on the date shown below, being deposited with the United States Postal Service with sufficient postage as first class mail in an envelope addressed to the Commissioner of Patents and Trademarks, Washington, D.C. 20231.

Name: Judith A. Beckstrand
Date: 24 Sep 2001 Signature: Judith A. Beckstrand

---

brief and fees therefor, are dealt with in the accompanying TRANSMITTAL OF APPEAL BRIEF.

This brief is transmitted in triplicate.

This brief contains these items under the following headings, and in the order set forth below:

I REAL PARTY INTEREST

II RELATED APPEALS AND INTERFERENCES

III STATUS OF CLAIMS

IV STATUS OF AMENDMENTS

V SUMMARY OF INVENTION

VI ISSUES

VII GROUPING OF CLAIMS

VIII ARGUMENTS REJECTIONS UNDER 35 U.S.C. 103

IX CLAIMS INVOLVED IN THE APPEAL

X OTHER MATERIALS THAT APPELLANT CONSIDERS NECESSARY OR DESIRABLE

The final page of this brief bears the practitioner's signature.

I REAL PARTY INTEREST

The real party in interest in this appeal is International Business Machines Corporation, Armonk, New York.

II RELATED APPEALS AND INTERFERENCES

No appeals or interferences will directly affect, or be directly affected by, or have a bearing on the Board's decision in this appeal.

III STATUS OF CLAIMS

A. TOTAL NUMBER OF CLAIMS IN APPLICATION

Claims in the application are: 1-8

B. STATUS OF ALL THE CLAIMS

1. Claims canceled: None

2. Claims withdrawn from consideration but not canceled: None

3. Claims pending: 1-8

4. Claims allowed: None

5. Claims rejected: 1-8

C. CLAIMS ON APPEAL

The claims on appeal are: 1-8

IV STATUS OF AMENDMENTS

The status of any amendment filed subsequent to the final rejection is, insofar as understood by appellant, as follows:

Pursuant to the communication from Examiner Majid Banankhah dated 5 July 2001, appellant's amendment filed 22 June 2001 will be entered upon the timely submission of a Notice of Appeal and this Appeal Brief with requisite fees.

V SUMMARY OF INVENTION

The invention provides a multi-tasking operating system for managing simultaneous access to scarce or serially re-usable resources by multiple process threads 100. A stationary queue 110, 112 (Page 5, lines 2-10; page 7, lines 3-10 and Table 8, page 15) allocates access to the resources amongst threads 102, 104 in fair (Page 16, lines 22-26), or FIFO order -- also referred to as "next thread in line" (Page 5, line 20).

The stationary queue is built on two counters, wait counter 110 (also referred to as the number_forced_to_wait NFW counter described at page 6 lines 12-18) and satisfied counter 112 (also referred to as the number_satisfied or NSC counter described at page 6 lines 20 to page 7 line 2) which are used by two code routines, sleep code module 120 (Page 5, lines 14-16) and wake up code module 130 (Page 5 lines 18-21) to generate a pair of identifiers -- block identifier 122, and run identifier 132. The use of counters 110, 112 and identifiers 122, 132 when adding a thread to and removing a thread from the stationary queue is illustrated in Figure 2 and described at page 8, lines 1-26. GetInLine() routine (Table 8) places a thread "in line" the

stationary queue. The thread uses this function to "take a number", and when the number is called, this thread awakens. There is no thrashing and FIFO order for service of waiting threads is assured (Page 16, lines 17-26).

VI ISSUES

Whether claims 1 through 8 are unpatentable under 35 U.S.C. 103 over Davidson et al. (U. S. Patent 5,630,136), in view of Periwal et al. (U. S. Patent 5,644,768).

VII GROUPING OF CLAIMS

Group I claim 1 stands alone.

Group II claims 2 and 3 stand or fall together.

Group III claims 4-7 stand or fall together.

Group IV claim 8 stands alone.

VIII ARGUMENTS

ARGUMENT: REJECTIONS UNDER 35 U.S.C. 103

Claims 1-8 have been rejected under 35 U.S.C. 103 over Davidson et al. (U. S. Patent 5,630,136), in view of Periwal et al. (U. S. Patent 5,644,768).

Appellant asserts that the Examiner has failed to make a prima facie showing of obvious by failure to show that either Davidson or Periwal or their combination teach critical concepts in the claims, as follows: (1) the use of a stationary queue (2) to assure FIFO or "fair" order.

It is an improper use of a reference to modify its structure.

Attention is directed to the case of In re Hummer, 241 F.2d 742, where it was held specifically:

> "Prior patent is a reference only for what it clearly discloses or suggests; it is an improper use of a patent as a reference to modify its structure to one it does not suggest."

The Examiner interprets Davidson's "serializing" and Periwal's mutex as teaching applicants "fairness"

The Examiner states:

"Applicant... argue 'On the other hand, applicant has provided a stationary queue. Such a queue, which includes two counters, insures not only fairness, but also on thread at a time is awakened and given access, so system performance in resource constrained scenarios is maintained. There is no thrashing of a run/wait queue in the kernel". In response it is submitted that, first fairness is interpreted as serializing threads, the reference of Davidson teaches of serializing threads in col. 2, lines 49-54... Regarding one thread at a time, Davidson teach 'However, at most one of the baton objects for a given multi threading unsafe resource can own the baton at a given time."

"...The data structure which causes the thread access to be serialized is taught by Davidson, and to ensure the serializability of thread access, Periwal teaches of mutex record." (Office Action, 25 April 2001, page 4. Emphasis added.)

Applicant traverses this characterization of the teachings of Davidson and Periwal.

Davidson defines "serial" as "mutually exclusive", not as "fair" or in "FIFO" order

Davidson defines what is meant by "serializing" as follows:

> "...a MUTEX lock must insure that accesses to the view object 22 are mutually exclusive, that is, serial." (Col. 5, lines 28-30.)

Serial, in the sense of mutually exclusive, is what Davidson is teaching, but is not what applicant is claiming. Applicant's claims are drawn to "in order of request" (claim 1), "FIFO order" (claims 2 and 3), "fair order" (claims 4-7), or "next... in line" (claim 8).

A mutex record does provide mutual exclusion among threads, but does not provide "fair" or "FIFO" order

Periwal (and also Davidson) relate to the OS MUTEX primitive for locking, or sleep/wake. However, the order of

acquisition of a mutex is not defined, and does not assure fair order.

> "By default, if multiple threads are waiting for a mutex, the order of acquisition is undefined." (See On-line UNIX manual pages at the University of Virginia [http:// www.cs.virginia.edu/cgi-bin/manpage?section=3T&topic=mutex], copy attached. Emphasis added)

Why is this order undefined? OS MUTEX primitives simultaneously wake all threads which are sleeping/waiting on a mutex-protected resource. It is then happenstance which thread actually makes it first to access the protected resource (such as by successfully obtaining Davidson's "baton"), and the others must go back to sleep/wait state. If, for example, 15 threads are waiting on a resource, as one becomes free, all 15 threads are awakened, one grabs the baton, and the other 14 all must return to wait state. This behavior causes thrashing on the multi-tasking OS's system "run" queue, where threads that are runnable or waiting to be runnable are tracked. By the time the last of the 15 threads is satisfied, it has make a lot of unsuccessful attempts to be serviced. A great deal of system resources is expended in managing such a run/wait operation.

Periwal teaches a mutex and a nesting mechanism for preventing deadlocks when accessing the mutex. This is done to prevent deadlock conditions but as described hereafter, as with Davidson, there is no teaching of the fair, or FIFO, ordering of threads.

The Examiner correctly observes that Davidson does fail to explicitly teach of a queue for allocating access to resources. In fact, Davidson and Periwal both fail to teach anything about enforcing order in the system. Periwal's reference to awakening "the next sleeping thread" (Col. 11, line 57) and to "subsequent threads or processes" (Col. 11, line 67) says nothing about enforcing order. No mention is made about how "next" or "subsequent" is determined, and in fact in Periwal (as well as in Davidson) that order is undefined, as is apparent from an examination of the code in the Source Code Appendix of Periwal.

In the Source Code Appendix, Periwal provides for awakening (which is done to allocate a resource to a next thread) at the following places for each of several operating systems:

Operating System	Col	Line	Description
Apollo	21	3	**mutex**_$unlock
OS/2	23	26	return DosRelease**Mutex**Sem
Posix	25	49	return pthread_**mutex**_lock
Solaris	29	4	return **mutex**_unlock
VMS	31	26	return ISC_**mutex**_unlock
Netware	37	16	return ISC_**mutex**_unlock
Windows	33	34	Leave**CriticalSection**

In each case, Periwal leaves the determination of which thread is to next have access to the "mutex" or "critical section" up to the operating system, and teaches no mechanism or process for assuring FIFO, fair, next thread in line, or in order of request.

In the case of a **mutex**, as taught in the on-line UNIX manual page at the URL noted above, "By default, if multiple threads are waiting for a mutex, the order of acquisition is undefined." (Emphasis added). By using a mutex structure in six of his seven operating system examples without any additional teaching with respect to FIFO, fair, etc. order, Periwal inherently teaches that the order of acquisition is undefined. Consequently, Periwal's system, as also Davidson's, necessarily involves the thrashing that applicant's invention avoids with a thread specific (single, next in fair or FIFO order) awakening.

In the case of a **critical section**, the seventh (in the

above list) of the seven examples in the source code appendix of Periwal, as taught in the Microsoft Visual Studio Version 6.0 Help function on "EnterCriticalSection" (copied into a note dated 06/20/2001 09:40 AM from Bill Wilhelm to Shelley Beckstrand, copy attached) for mutual exclusion processing, each thread desiring to enter a critical section must first request and obtain ownership. When a thread having ownership, releases it, it is available for another thread. However, there is no structure or method taught by Periwal or provided by the Operating System for assuring that the "next" or "subsequent" thread to obtain ownership of the critical section is the next in fair, or FIFO, order. It is merely the "next" of all competing threads to win the race for ownership of the critical section. Applicants avoid such a race, or thrashing, by awakening only one thread, the one thread which applicant claims is in "FIFO order", "fair order", "in order of request", or "next thread in line".

Applicant teaches and claims a "stationary queue" for assuring fairness

The Examiner asserts that "there is no teaching of any (FIFO) queue or queue-like behavior without the use of a real queue in the claims or even in the specification."

(Office Action, Paper No. 6, page 4.)

Applicant traverses. The stationary queue is described in the specification (See Section V, Summary of the Invention, supra), is explicitly claimed in claims 1-3, and elements of such a queue are present in the other claims, 4-8. This stationary queue, and the claimed elements of it, is provided to avoid the thrashing that is inherent in the MUTEX process and assure FIFO order, fair order, order of request, or next in line order.

A stationary queue is not a queue in the normal use of the term "queue", but rather a structure which presents queue-like behavior without being a queue. Appellant's invention achieves queue-like behavior (first-come, first-served) without the use of a real queue. As appellant explains in his specification:

> In accordance with this invention, the solution to this scarce resource management problem produces the same result as a queue or FIFO, but there is no memory actually associated with the queue or FIFO structure. No data or identifiers are moved into or out of this queue, hence the term 'stationary'. Only the read and write pointers of the queue need be maintained since

the data that would ordinarily be stored in the queue is impliedly the counter, or pointer, values themselves. (Page 5, lines 2-10.)

As the Examiner correctly observes, a queue is a data structure. However, as is apparent to those of skill in the art, it is a very specific data structure, requiring storage for all the possible elements that the queue may hold, and a multitude of methods for manipulating the queue, including Init, Empty, IsEmpty, Enqueue, Dequeue.

Appellant has provided a system and method which achieves queue-like behavior without any of the standard queue memory overhead. Appellant does not require memory for each element on the queue. In fact, storage requirements are the same regardless how many sleeping threads are to be placed in the stationary queue. Rather, appellant requires only two variables, or counters: the cumulative counter of threads forced to wait counter 112, and the cumulative number of threads that have been served counter 110. These counters 110, 112 are used to generate thread-specific wake-up calls which avoid the thrashing that is inherent in the MUTEX process relied upon by Davidson and Periwal. It is as though appellant had a queue, but without the memory and methods required to instantiate a typical

queue.

Applicant's claims distinguish Davidson and Periwal, and their combination

Each of appellant's claims distinguish Davidson, Periwal and any combination of them, as follows.

Claim 1, at lines 7-9, recites "a stationary queue for allocating access to said resource amongst said threads one-by-one in order of request".

Claim 1 is appellant's broadest claim, and its patentability rests on the argument that a stationary queue which is used to assure fair order is not taught by either Davidson or Periwal, nor by their combination.

Claim 2, at lines 9 and 10, specifies that "a next thread in line is to be re-animated and granted access in FIFO order."

This claim is specifically drawn to the sleep code routine for generating the block ID and the wake-up code for generating the run ID. These represent a specific embodiment of the system for implementing the stationary

queue.

Claim 3 depends from claim 2.

Claim 4 recites that a thread is awakened and granted access to a resource in fair order, and claims 5-7 recite that requesting threads are animated one-by-one in order of request, or in fair order.

Claim 8, recites the components of a stationary queue comprising the counter pair mentioned above and, at lines 10 and 11, "a wake-up code routine for generating a unique run identifier when a next thread in line is to be re-animated and granted access to said resource" and "said wake-up code routine being responsive to said satisfied counter for generating said run identifier" (emphasis added). This provides a "thread specific" wake up call, described in appellant's specification at page 8 lines 14-26 using the two counters that appellant provides to wake up only the next thread (as distinguished from waking up all waiting threads which must then contend for the MUTEX). The inherent result is no contention or thrashing for ownership of the resource. This is not taught by Davidson or Periwal.

IX CLAIMS INVOLVED IN THE APPEAL

1. A multi-tasking operating system for managing simultaneous access to scarce or serially re-usable resources by multiple process threads, comprising:

at least one resource;

a plurality of threads requesting access to said resource; and

a stationary queue for allocating access to said resource amongst said threads one-by-one in order of request.

2. A multi-tasking operating system stationary queue for managing simultaneous access to scarce or serially re-usable resources by multiple process threads, the stationary queue comprising:

a sleep code routine for generating a unique block identifier when a process thread temporarily cannot gain access to said resource and must be suspended; and

a wake-up code routine for generating a unique run identifier when a next thread in line is to be re-animated and granted access to said resource in FIFO order.

3. The system of claim 2, further comprising:

a wait counter for counting the cumulative number of threads that have been temporarily denied the resource; a satisfied counter for counting the cumulative number of threads that have been denied access and subsequently granted access to said resource;

said sleep code routine being responsive to said wait counter for generating said run identifier; and

said wake-up code routine being responsive to said satisfied counter for generating said run identifier.

4. A method for managing simultaneous access to scarce or serially re-usable resources by multiple process threads, comprising the steps of:

responsive to a request from a thread for a resource which is not available, creating a block identifier

based on the number of threads temporarily denied the resource; and

blocking said thread using said block identifier, thereby enabling subsequent wake up of said thread in fair order after said number of threads have been serviced.

5. A method for managing simultaneous access to scarce or serially re-usable resources by multiple process threads, comprising the steps of:

responsive to a resource becoming available, creating a run identifier based on the number of threads that have been first forced to wait and have been subsequently satisfied; and

awakening and running one next thread using said run identifier, thereby granting access to said resource by said process threads one-by-one in fair order.

6. A method for managing simultaneous access to scarce or serially re-usable resources by multiple process threads, comprising the steps of:

responsive to a request for a resource which is not available,

creating a block identifier based on the number of threads temporarily denied the resource; and

blocking the thread using said block identifier; and

responsive to a resource becoming available,

creating a run identifier based on the number of threads that have been first forced to wait and have been subsequently satisfied; and

awakening and running a next thread in fair order using said run identifier.

17. A memory device for storing signals for controlling the operation of a computer to manage simultaneous access to scarce or serially re-usable resources by multiple process threads, according to the steps of

responsive to a request for a resource which is not available,

creating a block identifier based on the number of threads temporarily denied the resource; and

blocking the thread using said block identifier; and

responsive to a resource becoming available,

creating a run identifier based on the number of threads that have been first forced to wait and have been subsequently satisfied; and

awakening and running the one next thread in fair order using said run identifier.

8. A memory device for storing signals to structure the components of a digital computer to form a stationary queue for managing simultaneous access to scarce or serially re-usable resources by multiple process threads, comprising:

a sleep code routine for generating a unique block identifier when a process thread temporarily cannot gain access to said resource and must be suspended;

a wake-up code routine for generating a unique run

identifier when a next thread in line is to be re-animated and granted access to said resource;

a wait counter for counting the cumulative number of threads that have been temporarily denied the resource;

a satisfied counter for counting the cumulative number of threads that have been denied access and subsequently granted access to said resource;

said sleep code routine being responsive to said wait counter for generating said run identifier; and

said wake-up code routine being responsive to said satisfied counter for generating said run identifier.

X. OTHER MATERIALS THAT APPELLANT CONSIDERS NECESSARY OR DESIRABLE

Attached are copies of the following materials referred to in the brief.

Microsoft Visual Studio Version 6.0 Help function on

"EnterCriticalSection" (copied into a note dated 06/20/2001 09:40 AM from Bill Wilhelm to Shelley Beckstrand)

On-line UNIX manual pages at the University of Virginia [http:// www.cs.virginia.edu/cgi-bin/manpage?section=3T&topic=mutex]

XI. CONCLUSION

Appellant urges that claims 1-8 be allowed.

Shelley M Beckstrand
Attorney for Appellant
Reg. No. 24,886

314 Main Street
Owego, NY 13827

Phone: (607) 687-9913
Fax: (607) 687-7848

Bill Wilhelm
06/20/2001 09:40 AM

To: shelley beckstrand
cc:
From: Bill Wilhelm/Endicott/IBM@IBMUS
Subject:

Excerpt from Microsoft Visual Studio Version 6.0 Help function on "EnterCriticalSection" function for mutual exclusion processing.

EnterCriticalSection
The EnterCriticalSection function waits for ownership of the specified critical section object. The function returns when the calling thread is granted ownership.

```
VOID EnterCriticalSection(
  LPCRITICAL_SECTION lpCriticalSection   // pointer to critical
                                         // section object
);
```

Parameters
lpCriticalSection
Pointer to the critical section object.
Return Values
This function does not return a value.

Remarks
The threads of a single process can use a critical section object for mutual-exclusion synchronization. The process is responsible for allocating the memory used by a critical section object, which it can do by declaring a variable of type CRITICAL_SECTION. Before using a critical section, some thread of the process must call the InitializeCriticalSection or InitializeCriticalSectionAndSpinCount function to initialize the object.

To enable mutually exclusive access to a shared resource, each thread calls the EnterCriticalSection or TryEnterCriticalSection function to request ownership of the critical section before executing any section of code that accesses the protected resource. The difference is that TryEnterCriticalSection returns immediately, regardless of whether it obtained ownership of the critical section, while EnterCriticalSection blocks until the thread can take ownership of the critical section. When it has finished executing the protected code, the thread uses the LeaveCriticalSection function to relinquish ownership, enabling another thread to become owner and access the protected resource. The thread must call LeaveCriticalSection once for each time that it entered the critical section. The thread enters the critical section each time EnterCriticalSection and TryEnterCriticalSection succeed.

Once a thread has ownership of a critical section, it can make additional calls to EnterCriticalSection or TryEnterCriticalSection without blocking its execution. This prevents a thread from deadlocking itself while waiting for a critical section that it already owns.

Any thread of the process can use the DeleteCriticalSection function to release the system resources that were allocated when the critical section object was initialized. After this function has been called, the critical section object can no longer be used for synchronization.

If a thread terminates while it has ownership of a critical section, the state of the critical section is undefined.

QuickInfo
Windows NT: Requires version 3.1 or later.
Windows: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winbase.h.
Import Library: Use kernel32.lib.

See Also
Synchronization Overview, Synchronization Functions, DeleteCriticalSection, InitializeCriticalSection, InitializeCriticalSectionAndSpinCount, LeaveCriticalSection TryEnterCriticalSection

G.W. Wilhelm, Jr.
Electronic Media Management System - Subsystem Development
(607)755-8943 t/l 855-8943
Internet: wilhelm@us.ibm.com

Bill Wilhelm
06/20/2001 09:51 AM

To: Shelley Beckstrand/Endicott/Contr/IBM
cc:
From: Bill Wilhelm/Endicott/IBM@IBMUS
Subject:

Shelley,
Check out the following URL:

http://www.cs.virginia.edu/cgi-bin/manpage?section=3T&topic=mutex

It is a link to some manual pages for UNIX from the University of Virgina. Note especially the NOTES section near the bottom of the Web page which I have quoted here below (emphasis mine):

"By default, if multiple threads are waiting for a mutex, the order of acquisition is **undefined**."

This same reference can be found in any on-line UNIX man pages (at Cornell, Princeton,...)

Bill

G.W. Wilhelm, Jr.
Electronic Media Management System - Subsystem Development
(607)755-8943 t/l 855-8943
Internet: wilhelm@us.ibm.com

NAME

mutex - concepts relating to mutual exclusion locks

DESCRIPTION

FUNCTION	ACTION
mutex_init	Initialize a mutex.
mutex_destroy	Destroy a mutex.
mutex_lock	Lock a mutex.
mutex_trylock	Attempt to lock a mutex.
mutex_unlock	Unlock a mutex.
pthread_mutex_init	Initialize a mutex.
pthread_mutex_destroy	Destroy a mutex.
pthread_mutex_lock	Lock a mutex.
pthread_mutex_trylock	Attempt to lock a mutex.
pthread_mutex_unlock	Unlock a mutex.

Mutual exclusion locks (mutexes) prevent multiple threads from simultaneously executing critical sections of code which access shared data (that is, mutexes are used to serialize the execution of threads). All mutexes must be global. A successful call to acquire a mutex will cause another thread that is also trying to lock the same mutex to block until the owner thread unlocks the mutex.

Mutexes can synchronize threads within the same process or in other processes. Mutexes can be used to synchronize threads between processes if the mutexes are allocated in writable memory and shared among the cooperating processes (see mmap(2)), and have been initialized for this task.

Initialization

Mutexes are either intra-process or inter-process, depending upon the argument passed implicitly or explicitly to the initialization of that mutex. A statically allocated mutex does not need to be explicitly initialized; by default, a statically allocated mutex is initialized with all zeros and its scope is set to be within the calling process.

For inter-process synchronization, a mutex needs to be allocated in memory shared between these processes. Since the memory for such a mutex must be allocated dynamically, the mutex needs to be explicitly initialized with the appropriate attribute that indicates inter-process use.

Locking and Unlocking

A critical section code is enclosed by a call lock the mutex and the call to unlock the mutex to protect it from simultaneous access by multiple threads. Only one thread at a time may possess mutually exclusive access to the critical section of code that is enclosed by the mutex-locking call and the mutex-unlocking call, whether the mutex's scope is intra-process or inter-process. A thread calling to lock the mutex either gets exclusive access to the code starting from the successful locking until its call to unlock the mutex, or it waits until the mutex is unlocked by the thread that locked it.

Mutexes have ownership, unlike semaphores. Only the thread that locked a mutex, (that is, the owner of the mutex), should unlock it.

If a thread waiting for a mutex receives a signal, upon return from the signal handler, the thread resumes waiting for the mutex as if there was no interrupt.

Caveats

Mutexes are almost like data - they can be embedded in data structures, files, dynamic or static memory, and so forth. Hence, they are easy to introduce into a program. However, too many mutexes can degrade performance and scalability of the application. Because too few mutexes can hinder the concurrency of the application, they should be introduced with care. Also, incorrect usage (such as recursive calls, or violation of locking order, and so forth) can lead to deadlocks, or worse, data inconsistencies.

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

SEE ALSO

mmap(2), shmop(2), mutex_destroy(3T), mutex_init(3T), mutex_lock(3T), mutex_trylock(3T), mutex_unlock(3T), pthread_mutex_destroy(3T), pthread_mutex_init(3T), pthread_mutex_lock(3T), pthread_mutex_trylock(3T), pthread_mutex_unlock(3T), pthread_create(3T), pthread_mutexattr_init(3T), attributes(5), standards(5)

NOTES

In the current implementation of threads, pthread_mutex_lock(), pthread_mutex_unlock(), mutex_lock() mutex_unlock(), pthread_mutex_trylock(), and mutex_trylock() do not validate the mutex type. Therefore, an uninitialized mutex or a mutex with an invalid type does not return EINVAL. Interfaces for mutexes with an invalid type have unspecified behavior.

By default, if multiple threads are waiting for a mutex, the order of acquisition is undefined.

USYNC_THREAD does not support multiple mapplings to the same logical synch object. If you need to mmap() a synch object to different locations within the same address space, then the synch object should be initialized as a shared object USYNC_PROCESS for Solaris, and PTHREAD_PROCESS_PRIVATE for POSIX.
